Lightweight Data-flow Analysis for Execution-driven Constraint Solving

Junaid Haroon Siddiqui
University of Texas at Austin
Austin, TX 78712
Email: jsiddiqui@utexas.edu

Darko Marinov
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Email: marinov@illinois.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712
Email: khurshid@ece.utexas.edu

***Abstract*—Constraint-based testing is a methodology for finding bugs in code, which has been successfully used for testing real systems. A key element of the methodology is generation of test inputs from *input constraints*, i.e., properties of desired inputs, which is performed by solving the constraints. We present a novel approach to optimize input generation from *imperative* constraints, i.e., constraints written as predicates in an imperative language. A well known technique for solving such constraints is execution-driven monitoring, where the given predicate is executed on candidate inputs to filter and prune invalid inputs, and generate valid ones. Our insight is that a lightweight static data-flow analysis of the given imperative constraint can enable more efficient solving. This paper describes an approach that embodies our insight and evaluates it using a suite of well-studied subject constraints. The experimental results show our approach provides substantial speedup over previous work.**

***Keywords*-Constraint-based testing, static data-flow analysis, multi-value comparisons**

I. Introduction

Constraint-based testing [5], [6], [13], [17], [21] is a methodology for finding bugs in code. A common application of the methodology is for black-box testing, for example, using *input constraints*, i.e., desired properties of inputs, and *oracle constraints*, i.e., expected properties of outputs and their relationship to inputs. However, constraint-based testing is also applicable for white-box testing, e.g., using symbolic execution [5], [17] where input constraints can be derived from program paths.

When used for black-box testing or for testing programs that require inputs that have specific properties, such as a valid XML document, a key requirement to apply constraint-based testing is to provide the constraints. Existing frameworks facilitate writing constraint by supporting different constraint languages, including declarative languages [15], which provide constructs for succinct formulation of complex properties but enforce the use of a likely unfamiliar programming paradigm, and imperative languages, such as C++, which likely offer familiarity but do not provide special constructs for writing constraints [25]. The focus of this paper is constraints written as imperative predicates in C++, which due to its wide familiarity provides the basis of a framework that can be used by many developers.

Given the constraints, a key technical challenge in automating this methodology is efficient generation of *valid* inputs, which satisfy the given constraints. For imperative constraints, it means generating inputs for which the corresponding predicate returns true. For programs that operate on dynamically allocated data with complex structural properties, input generation can require costly exploration of very large input spaces to find valid inputs, e.g., finding strings that represent valid XML documents.

The Korat framework [2] introduced a novel technique for solving imperative constraints – *execution-driven solving* – where a bounded space of candidate inputs is systematically explored by executing the given predicate on candidate inputs and monitoring the executions to filter and prune invalid candidates from the input space. This technique has been used effectively for finding bugs in a number of applications [10], [19], [27], [28] and similar techniques are the basis of other effective frameworks for systematic bug finding, e.g., lazy initialization in generalized symbolic execution [16] and the UC-KLEE framework [22]. While Korat enables an efficient way to prune the input space, it still requires checking each candidate input that is not pruned using a complete execution of the given predicate. However, such executions can be wasteful, particularly on candidate inputs that are largely similar.

This paper provides a novel approach for more efficient solving of imperative predicates using a lightweight static data-flow analysis. Our insight is that repeated predicate executions can be optimized by performing certain comparison operations, which determine the predicate's output, against *sets* of candidate values for fields used in the comparisons, i.e., performing *multi-value* comparisons, rather than comparing individual values in turn as in traditional Korat execution. Thus, predicate executions on many candidate inputs that are similar are *forwarded*, and the total execution cost is reduced. Conceptually, our approach resembles stateful model checking [29] where non-deterministic choice allows an expression to evaluate to different values, each of which is used in turn. However, a key difference is that we do not require storing and re-creating entire states. Moreover, our approach directly utilizes how field values determine the predicate's output in enumerating valid inputs as well as in pruning invalid ones.

```
class BST {
  struct Node {
    Node* left;
    Node* right;
    Node* parent;
    int data;
  };
  Node* root;
  int size;
 public:
  static Finitization* finitize(int size);
  bool repOk();
};
```

Figure 1. Binary search tree class definition

```
Finitization* BST::finitize(int c) {
  Finitization* f = Finitization::create<BST>();
  Domain<Node>* nodes = f->domain<Node>(c);
  f->set(&BST::root, nodes);
  f->set(&BST::size, f->domain<int>(c, c));
  f->set(&Node::left, nodes);
  f->set(&Node::right, nodes);
  f->set(&Node::parent, nodes);
  f->set(&Node::data, f->domain<int>(1, c));
  return f;
}
```

Figure 2. Finitization for Korat

We make the following contributions:

- **Lightweight static data-flow analysis for constraint solving.** We introduce the idea of utilizing def-use analysis in optimizing repeated predicate executions on similar inputs during Korat search.
- **Multi-value comparisons.** We introduce the idea of comparing sets of values with a desired value to compute the predicate's output on its future executions that are thus forwarded.
- **Implementation.** We embody our approach in a prototype tool that is available for download[1].
- **Evaluation.** We use a suite of subject constraints that have previously been studied in several projects and serve as a benchmark for evaluating constraint-solving for test-input generation. Experimental results show our approach provides a significant speed-up over Korat.

II. Background and Motivational Example

We consider a binary search tree and describe how its structural constraints are solved using the Korat algorithm to produce test inputs. We then describe how our technique based on multi-value comparisons using static data-flow analysis makes the constraint solving much faster while keeping the algorithm correct.

A. Example: Binary search tree

The binary search tree class (BST) is defined in Figure 1. It contains an inner class `Node` that represents a single node in the BST. The `Node` contains `left` and `right` pointers to other nodes, a `parent` pointer, and an integer `data` field. The `BST` class contains a pointer to the `root` node and the number of nodes in the tree in the `size` field. Two methods must be provided for constraint solving with Korat: a `finitize` method that describes bounds for analysis (called *finitization*) and a `repOk` predicate method that tells if a particular instance of BST is *valid* or not (also called the *class invariant*).

Figure 2 shows an example finitization for BST. We create a domain of `Node` objects and require `root`, `left`, `right`, and `parent` to take values only from this domain. For `data` and `size` fields, we create integer domains of appropriate sizes.

```
bool BST::repOk() {
  set<Node*> visited;
  stack<tuple<Node*, int, int> > wl;
  if (root) {
    wl.push(make_tuple(root,
    numeric_limits<int>::min(),
    numeric_limits<int>::max()));
    visited.insert(root);
  }
  while (!wl.empty()) {
    Node* c = get<0>(wl.top());
    int min = get<1>(wl.top());
    int max = get<2>(wl.top());
    wl.pop();
    if (c->data < min || c->data > max)
      return false;
    if (c->left) {
      if (!visited.insert(c->left).second)
        return false;
      if (c->left->parent != c)
        return false;
      wl.push(make_tuple(c->left, min, c->data-1));
    }
    if (c->right) {
      if (!visited.insert(c->right).second)
        return false;
      if (c->right->parent != c)
        return false;
      wl.push(make_tuple(c->right, c->data+1, max));
    }
  }
  return size == visited.size();
}
```

Figure 3. Class invariant for binary search tree

Next, we provide the `repOk` function for BST in Figure 3. It is the class invariant and checks the BST properties. These properties are: (1) acyclicity along `left` and `right` pointers, (2) correct `parent` pointers, and (3) larger `data` values are stored in right sub-tree while smaller `data` values are stored in left sub-tree. The given function checks these properties using a work-list based algorithm.

[1]http://svvat.ece.utexas.edu/tools/multivaluekorat

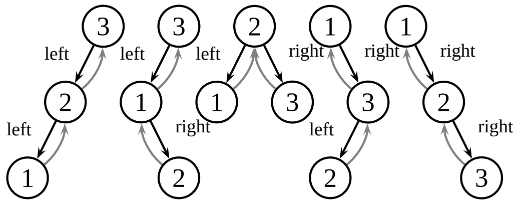


Figure 4. Binary search trees of 3 nodes with parent pointers

B. Traditional Korat

Korat is an algorithm for execution-driven constraint solving. It takes as input a class definition with its class invariant and a finitization function (provided by `repOk` and `finitize` methods in the above example).

To start solving the constraint, the Korat algorithm forms an initial candidate structure to test. The candidate is formed by assigning every field the first value from its domain specified in the finitization. Korat executes `repOk` on this candidate to check if it is valid.

During `repOk` execution, Korat monitors field accesses and builds a *field-access list*. After `repOk` finishes, it picks a new value for the last accessed field from its field domain and runs `repOk` again. If there are no more values in its field domain, it backtracks to the field accessed before it. This way Korat explores the state-space without testing every possible combination of values in the field domains.

Korat produces non-isomorphic inputs. Non-isomorphic inputs differ only in the identity of objects used and provide no additional fault-finding ability in testing code. To produce non-isomorphic inputs, Korat records the values of reference fields that are accessed by other reference fields of the same type. It only backtracks to `null`, values also referenced by other fields of the same type, and one new value. For example, if N_0 and N_1 are used by `root` and `left` pointers respectively, the `right` pointer will take a value from $\{\texttt{null}, N_0, N_1, N_2\}$. Choosing another value N_3 would form a structure isomorphic to the one formed using N_2.

Given the class definition, finitization, and class invariant, our desired output is a set of all concrete structures of a given size. For size 3, these structures are shown in Figure 4. We expect the constraint solver to list all of the five structures and only these five structures.

Korat starts its search from an empty tree with `root=null` and backtracks on accessed fields to try other values. To explain the working of Korat we describe its progression between two valid candidates shown at the right and left extremes of Figure 5.

When Korat analyzes Figure 5(a) using `repOk`, the fields accessed are `(root,` N_0`.data,` N_0`.left,` N_0`.right,` N_1`.parent,` N_1`.data,` N_1`.left,` N_1`.right,` N_2`.parent,` N_2`.data,` N_2`.left,` N_2`.right, size)`. After marking this as a valid candidate, it backtracks to N_2`.right` as `size` is already at its maximum value. All choices for N_2`.right` result in cyclic structures, so Korat backtracks to N_2`.left` which also results in cyclic structures. After each of these executions, Korat uses the field-access list generated as a result of the last execution. Since we assume that the `repOk` function is deterministic, the initial part of the field access list is the same.

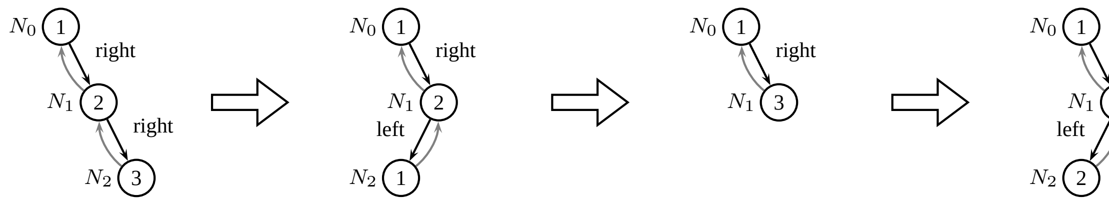
After backtracking past other fields, Korat resets N_1`.right` to `null` and backtracks to try other values for N_1`.left`. When it tries N_1`.left=`N_2, it fails because N_2`.parent=null` (its initial value). However the field access list has a new field and Korat tries its other values. N_2`.parent=`N_1 works but `repOk` fails because N_2`.data`$\not>$ 1. This is Figure 5(b). In total, Korat performs 16 `repOk` executions between these two states. Korat proceeds this way until all values are exhausted and finds all valid structures within the given bounds.

C. Korat$_{\text{multi}}$: multi-value comparisons

Korat$_{\text{multi}}$ uses multi-value comparisons based on lightweight data analysis. It is a technique for reducing the amount of `repOk` executions required to find all valid candidates by Korat. It is an automated technique that requires no modification to the `repOk` predicate.

Korat$_{\text{multi}}$ performs a static data-flow analysis on the LLVM bit-code [1] of the program and instruments it. This analysis identifies any field accesses in `repOk` that are used in a comparison that *directly results* in the predicate failing or succeeding. Here, "directly result" means that the result of the comparison is either returned directly or it is used in a conditional branch that results in returning from the function on either the true side or the false side. Once such comparisons are identified, we instrument them with a special call with the field accessed, the comparison, and the other side of the comparison as arguments. This special call performs a multi-value comparison against all values in the field domain of this field. If, however, the field access is not used in such a comparison, we instrument it with a simple call to monitor field access (like traditional Korat).

We explain the multi-value comparisons in Korat$_{\text{multi}}$ that are performed using this static data-flow analysis for the state of exploration shown in Figure 5(a). To repeat, the accessed fields are `(root,` N_0`.data,` N_0`.left,` N_0`.right,` N_1`.parent,` N_1`.data,` N_1`.left,` N_1`.right,` N_2`.parent,` N_2`.data,` N_2`.left,` N_2`.right, size)`. Like traditional Korat, Korat$_{\text{multi}}$ backtracks over the first few fields until it reset N_1`.right` to `null` and sets N_1`.left=`N_2. At this point, unlike traditional Korat, the `repOk` execution in Korat$_{\text{multi}}$ will not fail because N_2`.parent=null` (its initial value). The access to N_2`.parent` would have been statically instrumented with a function call. This function call is invoked during `repOk` execution and this is the first access to N_2`.parent`. Therefore, our optimization is



field+condition	values to try	field+condition	values to try	field+condition	values to try	field+condition	values to try
root	0,[N_0]	root	0,[N_0]	root	0,[N_0]	root	0,[N_0]
N_0.data	[1],2,3	N_0.data	[1],2,3	N_0.data	[1],2,3	N_0.data	[1],2,3
N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1	N_0.left	[0],N_0,N_1
N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]	N_0.right	0,N_0,[N_1]
N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~	N_1.parent=N_0	~~0~~,[N_0],~~N_1,N_2~~
1<N_1.data	~~1~~,[2],3	1<N_1.data	~~1~~,[2],3	1<N_1.data	~~1~~,2,[3]	1<N_1.data	~~1~~,2,[3]
N_1.left	[0],N_0,N_1,N_2	N_1.left	0,N_0,N_1,[N_2]	N_1.left	[0],N_0,N_1,N_2	N_1.left	0,N_0,N_1,[N_2]
N_1.right	0,N_0,N_1,[N_2]	N_1.right	[0],N_0,N_1,N_2	N_1.right	[0],N_0,N_1,N_2	N_1.right	[0],N_0,N_1,N_2
N_2.parent=N_1	~~0,N_0~~,[N_1],~~N_2~~	N_2.parent=N_1	~~0,N_0~~,[N_1],~~N_2~~	size=2	~~3~~	N_2.parent=N_1	[N_1]
2<N_2.data	~~1,2~~,[3]	1<N_2.data<2	~~1,2,3~~,			1<N_2.data<3	~~1~~,[2],~~3~~
N_2.left	[0],N_0,N_1,N_2					N_2.left	[0],N_0,N_1,N_2
N_2.right	[0],N_0,N_1,N_2					N_2.right	[0],N_0,N_1,N_2
size=3	[3]					size=3	[3]

Figure 5. Four intermediate states when using Korat with multi-value comparisons for binary search trees of three nodes. All accessed fields are shown in "field+condition" column along with any condition deduced from static light-weight analysis. For each field, the values to be tried are shown in "values to try" column. Striked out values are "forwarded" based on the comparison. Current value is shown in square brackets.

applicable, and Korat$_{\text{multi}}$ can do a multi-value comparison. It also gets as arguments the comparison (equals) and the value to compare against (`null` in this case). It does a multi-value comparison of this value against all values in the field domain of N_2`.parent`. All invalid choices for N_2`.parent` (that would result in returning `null`) are forwarded over without backtracking and executing `repOk` on them again. The only valid choice is used for further execution, and it reaches Figure 5(b). This takes 11 `repOk` executions instead of 16 for traditional Korat.

From there, Korat$_{\text{multi}}$ forwards over N_2`.data` and N_2`.parent` fields to reach Figure 5(c). Lastly, it backtracks and tries all fields while forwarding over `parent` and `data` fields to arrive at Figure 5(d).

The instrumentation based on static data-flow analysis and dynamic multi-value comparisons in Korat$_{\text{multi}}$ enable it to significantly reduce the number of `repOk` executions required while still producing all valid structures. For a binary search tree with 3 nodes, traditional Korat finds all 5 structures in Figure 4 using 238 `repOk` executions. In contrast, Korat$_{\text{multi}}$ requires only 173 `repOk` executions.

III. Technique

In this section, we go over all the algorithms we employ to enable multi-value comparisons in Korat$_{\text{multi}}$. We (1) describe the high-level Korat$_{\text{multi}}$ algorithm and define *marked* and *unmarked* candidates, (2) explain the multi-value comparisons of a value against all values in a field domain that *mark* candidates, (3) describe the data-flow analysis that instruments the `repOk` predicate with calls to multi-value comparisons, and (4) discuss the correctness with respect to the traditional Korat algorithm.

A. The Korat$_{multi}$ Algorithm

The Korat$_{\text{multi}}$ algorithm builds upon the Korat algorithm by utilizing the information that some candidates are *marked*. A candidate is *marked* when the result of running `repOk` can be determined without running `repOk`. This determination comes from a combination of a multi-value comparison (Section III-B) on the last accessed field and light-weight static data-flow analysis (Section III-C). The data-flow analysis determines the correlation of the return value of the `repOk` predicate and the result of the multi-value comparison.

Traditional Korat, after completing a `repOk` iteration and receiving field-access list, picks the last accessed field and chooses the next value from its field domain. On the other hand, Korat$_{\text{multi}}$ picks the next *unmarked* value from its domain. Marked values are either "known to succeed" meaning `repOk` *will* accept them or "known to fail" meaning `repOk` *will* reject them. Known to succeed values are included in successful candidates directly, while known to fail values are simply forwarded over. If there are no more unmarked values in the field domain, Korat$_{\text{multi}}$ backtracks to the field accessed before the last accessed field while clearing all markings on the last accessed field. This is done because the markings are only valid on one path.

Thus Korat$_{\text{multi}}$ divides candidates into marked and unmarked candidates. Unmarked candidates need a complete `repOk` execution, whereas marked candidates can be accepted or forwarded-over without executing `repOk`. This forwarding results in much fewer `repOk` executions and a substantially lower execution time per candidate considered and makes Korat$_{\text{multi}}$ much more efficient that traditional Korat.

B. Multi-value comparisons

Korat$_{\text{multi}}$ depends on candidates being *marked* during the execution of `repOk` predicate. These markings are done by multi-value comparisons. A multi-value comparison compares a given value against all values in the field domain of a field not yet accessed in the `repOk` predicate. If a field has already been accessed, the given value can only be compared against its assigned value and no multi-value comparison can take place.

A multi-value comparison is *only* useful if at least one of the possible results (`true` or `false`) can determine what the `repOk` predicate will return. This information is gathered statically (Section III-C).

The method that performs a multi-value comparison and determines if some candidate needs to be marked helps in forwarding candidates without running `repOk`, and we thus call it `forwardFn`. The `forwardFn` method takes five arguments. Three of these arguments are for the multi-value comparison. They are (1) the field whose field domain gives the values that are all compared against a given value, (2) the comparison operator (==, <, > etc.), and (3) the given value to be compared (e.g. `2` in `if (size==2)`). Two other boolean arguments are statically determined (Section III-C) and inform if a `true` result of the comparison means that `repOk` will return `true`, and if a `false` result of the comparison means that `repOk` will return `false`. For example, in "`if (size==2) return false;`", we *cannot* mark a candidate if the comparison results in a `true` value, but we *can* mark it to be forwarded if it results in a `false` value.

Algorithm 1 shows `forwardFn` and `useFn` methods, where the `useFn` method is invoked for every field access in traditional Korat and it builds the field-access list. We also use it in Korat$_{\text{multi}}$ to instrument all accesses except those that classify as *forward-able* and are thus instrumented with `forwardFn`.

The `forwardFn` method checks if the given field has not been accessed before (like `useFn`). If not, it performs a multi-value comparison against all values in the field domain of this field. If the comparison is successful and `trueForward=true` (true side leads to returning constant `true`), it marks those values as accepted without even running `repOk` on them. Similarly, if the comparison is false and `falseForward=true` (false side leads to returning constant `false`), it marks them as skipped. If the initial value (as Korat initializes all fields to the first value in their field domains) of this field is marked as accepted or skipped during this process, it *forwards* to the first unmarked (not accepted, not skipped) value. If all values are marked, it chooses the last value and proceeds. These markings are used by the Korat$_{\text{multi}}$ algorithm to forward candidates without running `repOk`.

Algorithm 1 Algorithm for dynamic access monitoring

```
 1: function USEFN(Variable v)
 2:     if v is a controlled variable and not accessed before then
 3:         add v to field-access list
 4:     end if
 5: end function

 6: function FORWARDFN(Variable v, Cond c, Value otherVal,
    trueForward, falseForward)
 7:     if v is a controlled variable and not accessed before then
 8:         for all Values i in domain of v do
 9:             if result of applying c on i and otherVal is true then
10:                 if trueForward then
11:                     mark i as accepted without running repOk
12:                 end if
13:             else if falseForward then
14:                 mark i to be skipped (no need to run repOk)
15:             end if
16:         end for
17:         initialize v to first unmarked value
18:     end if
19:     return result of applying c on current value of v and
    otherVal
20: end function
```

Algorithm 2 Algorithm for traditional Korat instrumentation

```
1: for all instruction i in repOk do
2:     if i is a load instruction then
3:         insert call to useFn before i
4:     end if
5: end for
```

C. Data-flow analysis

Traditional Korat monitors all field accesses made during `repOk` execution. Algorithm 2 shows monitoring and instrumentation of field accesses using LLVM [1]. The function `useFn` dynamically determines if this is the first access to this field, in which case it is added to the field-access list.

Korat$_{\text{multi}}$ performs a more extensive analysis of the function and instruments `load` instructions that meet a set of conditions with a call to the `forwardFn` function. If the conditions are not met, Korat$_{\text{multi}}$ instruments with the same `useFn` function.

There are four sets of conditions for `load` instructions to be instrumented with `forwardFn`. Algorithm 3 gives the core algorithm using one condition for clarity, while Table I describes the other three conditions. Algorithm 3 iterates over all `load` instructions in `repOk`. Each `load` instruction defines a new variable and provides a starting point for def-use analysis.

Korat$_{\text{multi}}$ performs simple def-use analysis on this variable traversing the `uses` list provided by LLVM and checks if it is eventually (e.g., after sign-extending or bit-truncation) *only* used in a comparison instruction (`icmp`). This means that the accessed value is directly used as one argument of a comparison. It then follows the `uses` list of the result of the `icmp` instruction and sees if it is eventually used *only* in

Algorithm 3 Algorithm for light-weight def-use analysis

```
for all instruction i in repOk do
    if i is a load instruction then
        j = followUseChain(i)
        if j is an icmp instruction then
            k = followUseChain(j)
            if k is a br instruction then
                t = leadsToConstRet(true block of br)
                f = leadsToConstRet(false block of br)
                if t or f then
                    replace icmp with a call to forwardFn
                    continue // go to next i
                end if
            end if
        end if
        insert call to useFn before load
    end if
end for
```

a conditional branch instruction (`br`). This means that the comparison is used inside an `if` statement. Next, it inspects the true and false *basic blocks* (also called *then* and *else* basic blocks) that the branch leads to. Each basic block is a set of sequentially executed instructions ending with a terminating control flow instruction. If the terminating instruction is a return instruction (`ret`), Korat$_{\text{multi}}$ determines if a constant is returned. If a constant is returned in either the true basic block, or the false basic block, or both, it replaces the comparison with a call to `forwardFn` with the original `icmp` operands as arguments to the function, along with boolean parameters determining which basic blocks lead to returning constant.

While inserting the call to `forwardFn`, Korat$_{\text{multi}}$ ensures that the true, respectively false, side of the comparison leads to the function returning `true`, respectively `false`. If not, it inverts the comparison for passing it to `forwardFn` and again inverts the return value from `forwardFn`. This simplifies the operation of `forwardFn`.

This high-level description skims over two important details: (1) following the `uses` lists and (2) determining if a basic block results in returning a constant.

Algorithm 4 describes the def-use analysis. To follow from one instruction to the next, it ensures that the target instruction is the only instruction that uses the result of the source instruction. If more than one instruction uses the result of the source instruction, it does not attempt to determine if the other use does not influence the branch it will later take and thus does not consider such a case. If the target instruction is a cast instruction (truncation, zero extending, or sign extending), it repeats the algorithm to find the instruction that uses the result of the cast. Cast instructions are common in LLVM because it is a typed language with no implicit type conversions.

Algorithm 5 gives the algorithm for determining if a basic block leads to a constant return. This works similar

Algorithm 4 Algorithm for following def-use chain

```
function FOLLOWUSECHAIN(instruction i)
    if result of i has one use then
        j = only use of result of i
        if j is a cast instruction then
            return followUseChain(j)
        end if
    end if
    return null
end function
```

to constant propagation, except that the value to propagate (the result of comparison) is not really constant. Assuming that the result of comparison is known (`true` or `false`), it analyzes if this result could have been propagated to the return instruction. Sometimes it can be propagated for a `true` result of the comparison, or a `false` result, or for both. This information is then used by `forwardFn` to mark candidates after performing a multi-value comparison.

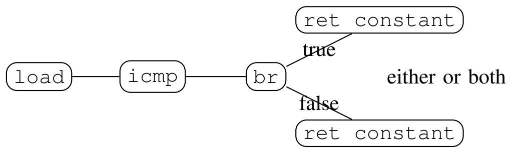
The algorithm is used on both the true and false target basic blocks of a conditional branch instruction to determine if either side results in returning a constant. The function works by considering the last instruction in the basic block (the only control flow instruction). If it is an unconditional branch to another basic block, it recursively invokes the same function on the target of the unconditional branch. If, however, the terminating instruction is a return instruction (`ret`), it checks the value returned. The returned value can be (1) a constant, (2) result of another instruction, or (3) a *phi constant*. A phi constant is a map from basic blocks to values where the value picked is based on the last basic block it was executing before a control flow instruction jumped into this block. The value corresponding to a basic block is again one of (1) a constant, (2) result of an instruction, or (3) another phi constant in the source basic block. Since it knows the basic block chain from the first `load` instruction to this `ret` instruction, it recursively resolves the phi constants. After this analysis, the algorithm may still be unable to resolve it, as it may depend on which basic block it came from before hitting the `load` instruction. When the returned value is a constant or a phi constant that it resolved to a constant, the algorithm returns this value.

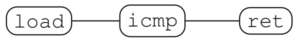
Note that the def-use analysis is light-weight because it only attempts to find the *last* use of some field before every return. It also does not need to consider earlier `load` instructions accessing the same field. This would have been a consideration if it had to make decisions statically. However, the def-use analysis delays the decision making until it actually executes the `repOk` predicate. At this time, it can monitor if a particular field is accessed for the first time or not (like traditional Korat does) and uses this information to enable forwarding for *that* access.

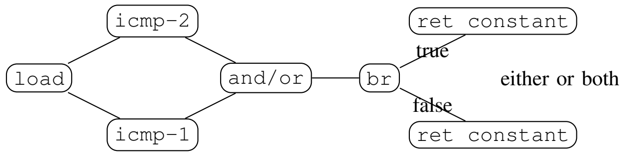
Table I shows all four conditions in which we instrument. Only the first condition is used in Algorithm 3 to describe

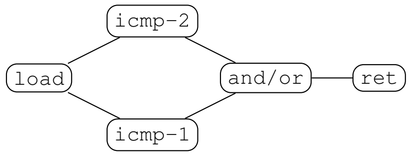
Table I
DEF-USE ANALYSIS OF `LOAD` INSTRUCTIONS WE INSTRUMENT.

No.	Use-chain	Description

1	 load
	icmp
	br
	true
	false
	ret constant
	ret constant
	either or both
		A `load` instruction defines a variable only used by an `icmp` instruction, the result of which is only used by a conditional `br` instruction leading to constant return on at least one of true and false sides.
		
		Example:
		`if(size!=visited.size())`
		`return false;`

2		A `load` instruction defines a variable only used by an `icmp` instruction, the result of which is used by a `ret` instruction. Example: `return size!=visited.size();`

3	 load
	icmp-2
	icmp-1
	and/or
	br
	true
	false
	ret constant
	ret constant
	either or both
		A `load` instruction defines a variable only used by two `icmp` instructions, where the results of both instructions are only used by an `and` or an `or` instruction, whose result is used by a conditional `br` instruction leading to constant return on at least one of true and false sides.
		
		Example:
		`if (data < min || data > max)`
		`return false;`

4		A `load` instruction defines a variable only used by two `icmp` instructions, where the results of both instructions are only used by an `and` or an `or` instruction, whose result is only used by a `ret` instruction. Example: `return data < min || data > max;`

Algorithm 5 Algorithm to see if a block leads to a constant return

```
 1: function LEADSTOCONSTANTRET(BasicBlock b)
 2:     i = last instruction in block b
 3:     if i is ret instruction then
 4:         v = value returned by i after resolving phi nodes
 5:         if v is constant then
 6:             return v
 7:         end if
 8:     else if i is an unconditional branch then
 9:         return leadsToConstantRet(target of i)
10:     end if
11:     return null
12: end function
```

the core concepts. The second condition is when the result of a comparison is directly returned without being used in a branch instruction. The next two cases are when two comparisons are made on the same field with an AND or an OR operation joining them. In such cases the variable defined by a `load` instruction is used in two `icmp` instructions and their results are used in an `and` or an `or` instruction whose result is used in a branch leading to a return (case 3) or directly returned (case 4). This can be easily generalized to multiple comparisons which are then joined by `and` or `or` instructions. However, our current implementation is limited to two comparisons.

Other limitations of our current implementation include instrumenting only one function (`repOk`). Any called helper functions are not instrumented. Additionally, we only support integer comparisons. These are, however, not fundamental limitations of the algorithm and more a matter of defining the scope of the implementation.

D. Correctness

For correctness, note that the candidates considered by Korat$_{\text{multi}}$ are the same as those considered by traditional Korat. However, Korat$_{\text{multi}}$ divides the candidates into unmarked (complete `repOk` execution) and marked (identified during `forwardFn` as accepted or rejected without running `repOk` again) and the union of marked and unmarked candidates is the same as traditional Korat. Thus, it suffices to show that marked candidates are correctly classified as accepted or rejected.

For marked candidates, we divide `repOk` into two parts. The first part goes from the start of `repOk` to the first `forwardFn` call and the second part from the `forwardFn` call to the `ret` statement.

We determined using static data-flow analysis that a `true` or a `false` return from `forwardFn` would lead to a `true` or a `false` return from `repOk`. Thus we do not need to execute the second part once we know the return value from `forwardFn`.

On the other hand, for every marked candidate we *do* have an execution of the first part. That execution touched everything except the last field, invoked the instrumented `forwardFn`, and marked candidates based on the comparison of the last accessed field. This execution is shared by all candidates who only vary in this last accessed field. Thus we have a dynamic execution of the first part (shared by more than one candidate) and a static knowledge of the behavior of the second part. Hence, Korat$_{\text{multi}}$ generates the same set of valid inputs as the traditional Korat algorithm.

IV. Evaluation

We evaluate Korat$_{\text{multi}}$ using two metrics: the number of `repOk` executions and the time it takes to generate valid inputs.

We use five complex structures to evaluate our technique. For each structure, we consider five different sizes. For each example, we generate structures of exactly the given size with unique elements. Our experiments were run on a machine with two Intel Xeon 2.93GHz 6-core processors and 24GB of memory.

To instrument a structure whose definition (along with `repOk` and `finitize` methods) is given in, say, `struct.cc`, we use the following command:

```
  llvm-g++ --emit-llvm --no-exceptions -c
struct.cc -o struct.o && llvm-ld -disable-inlining
-disable-internalize struct.o korat.o -o korat &&
opt -load forward.so -forward struct.bc | opt -O3
| llc -o struct.S && g++ struct.S -o struct
```

The command performs the following steps: (1) translate user code in `struct.cc` to LLVM bit code, (2) combine the user code with the Korat algorithm in `korat.o`, (3) apply the LLVM analysis in shared library `forward.so` required by multi-value comparisons, (4) optimize instrumented code (inlining etc.), (5) convert to native assembly, and (6) compile to native binary.

The structures we chose to test are min heap, dynamic order statistics, binary search tree, red-black tree, and sorted doubly linked list. Red-black trees are height balanced binary search trees using node colors and restrictions on assignment of that color. Dynamic order statistics are red-black trees where the nodes are further augmented with the size of the sub-tree rooted there. The problem of order statistics is concerned with returning the k^{th} smallest number in a set. For example, the minimum element in a set of n elements is the first order statistic while the maximum is the n^{th} order statistic. Dynamic order statistics enable retrieving any order statistic in logarithmic time. These structures provide a basis for developing more complex software and have been used in evaluating other software analysis techniques including the traditional Korat algorithm [2].

The results of our experiments are given in Table II. Our speedup ranges from 1.6X to 4.7X. This is shown graphically in Figure 6.

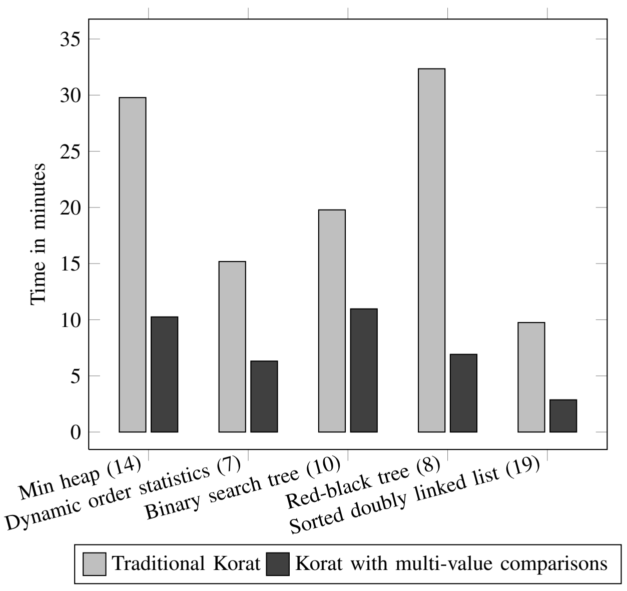


Figure 6. Time taken by Korat with and without multi-value comparisons. Number is parenthesis is the size of structures generated.

V. Related Work

Recent frameworks based on symbolic/concrete (aka concolic or dynamic symbolic) execution [3], [12], [23] that handle references/pointers are most closely related to Korat. A major difference is Korat's spirit of bounded *exhaustive* generation and backtracking based on last field accessed and not last branch taken. Generalized symbolic execution [16] follows Korat's spirit: lazy initialization of references has exactly the same effect as Korat's monitoring–both approaches consider the same candidates in the same order and generate the same structures. Practically, Korat is much faster since it is a specialized implementation–baseline Korat is an order of magnitude faster than a highly optimized version of lazy initialization on Java PathFinder [11]. In previous experiments using CUTE [23] for structural constraint solving, Korat outperformed CUTE by two orders of magnitude [24]. This is because of the overhead to keep symbolic state and Korat's specialized nature to backtrack on last accessed field. More recently, lazy initialization has also been implemented for equivalence checking of operations on complex structures in UC-KLEE [22].

Table II
COMPARISON OF KORAT$_{\text{MULTI}}$ WITH TRADITIONAL KORAT ALGORITHM FOR STRUCTURAL CONSTRAINT SOLVING

Subject	Size	Valid Structures	Traditional Korat Explored	Traditional Korat Time [s]	Korat with multi-value comparisons Explored	Korat with multi-value comparisons Time [s]	Speedup
Min heap	9	896	64,401	0.14	19,781	0.04	3.5X
	10	3,360	316,369	0.85	94,538	0.26	3.3X
	12	79,200	9,277,511	34.74	2,961,691	11.50	3.0X
	13	506,880	55,005,301	4:04.25	18,545,942	1:23.24	2.9X
	14	2,745,600	356,649,476	29:47.13	120,077,299	10:15.61	2.9X
Dynamic order statistics	3	2	3,356	0.02	1,654	0.01	2.0X
	4	4	42,294	0.44	20,115	0.21	2.1X
	5	8	415,922	6.74	188,321	3.06	2.2X
	6	16	3,646,604	1:27.68	1,558,574	37.24	2.4X
	7	33	28,564,440	15:11.70	11,502,100	6:19.81	2.4X
Binary search tree	6	132	49,524	0.46	30,469	0.28	1.6X
	7	429	279,427	3.50	166,762	2.10	1.7X
	8	1,430	1,555,219	25.50	906,048	14.77	1.7X
	9	4,862	8,562,721	2:55.95	4,891,974	1:39.93	1.8X
	10	16,796	46,729,370	19:47.34	26,269,077	10:58.52	1.8X
Red-black tree	4	4	20,482	0.19	8,397	0.08	2.4X
	5	8	161,122	2.40	53,956	0.79	3.0X
	6	16	1,259,268	26.83	360,500	7.59	3.6X
	7	33	7,962,572	3:52.97	1,938,263	55.66	4.2X
	8	56	51,242,194	32:21.49	11,077,150	6:55.75	4.7X
Sorted doubly linked list	15	1	1015748	19.20	294,897	5.88	3.3X
	16	1	2,162,624	44.75	622,576	13.83	3.2X
	17	1	4,587,452	1:49.12	1,310,703	32.65	3.3X
	18	1	9,699,256	4:12.08	2,752,494	1:14.80	3.4X
	19	1	20,447,156	9:45.19	5,767,149	2:52.19	3.4X
					Averge speedup		2.9X

SAT-based static analysis tools (Alloy [14], CBMC [4]) can perform bounded exhaustive checking for heap-allocated data. However, they require a translation of the whole program *and* its specification to a SAT formula: for non-small programs the formulas can choke the solvers. Korat requires solving only for input constraints, which are much simpler than the cumulative constraint that represents the correctness of the program under test. Static analysis tools that perform sound analysis of heap-allocated data (traditional shape analysis, verification conditions, separation logic) require more manual effort (in the form of loop invariants, additional predicates etc.) and have not been shown to scale to checking applications, which Korat readily handles.

UDITA [10] is a language that provides the ability to combine declarative and imperative predicates. It is based on JPF and delayed choice, which is an extension of the lazy initialization algorithm.

Dedicated generators in Korat [18] are most closely related to our optimization technique. Dedicated generators exploit common input properties to efficiently generate inputs. Basic generators support various properties: (1) if a value is in a set, (2) if two values are not equal, (3) if two values are equal, (4) if a value is smaller/greater than another value etc. More complex generators support if a pointer points to a tree or an acyclic graph etc. If the user takes the time to use the generator library, dedicated generators can be more efficient than our technique. However, for unmodified predicates, dedicated generators are not applicable, whereas our optimization technique can still apply. In fact, the `forwardFn` we introduced is a form of a dedicated generator which is introduced automatically where applicable.

Glass-box testing [7] uses the method under test to prune Korat's generation. Thus it takes a step further from the pure black-box approach of Korat. Glass-box testing can be optimized using our technique. Efficient backtracking [9] optimizes Korat by using abstract undo operations that enable re-using partial `repOk` executions. However, it needs explicit support from the `repOk` writer in the form of using un-doable operations. STARC [8] uses the Korat algorithm to repair structures. Our approach would make STARC efficient by forwarding over many invalid choices and thus reducing the number of `repOk` executions.

Korat has been parallelized for efficient analysis using pre-defined static partitions [20] and using a load-balancing approach [25]. Korat has also been combined with a symbolic execution engine [16] so that integer constraints can be solved without running `repOk` many times. However, the overhead of running an external symbolic execution engine is significant. Despite this, there are performance gains for non-reference fields. Also, the number of candidates generated is fewer as symbolic execution engine only generates a single solution for a given constraint. Another similar effort was Focused Generation for Korat [26]. Both of these works aimed at reducing the number of test inputs. However, this paper focuses on reducing the number of `repOk` executions to produce *all* test inputs.

VI. Conclusions and future work

We presented a novel approach to optimize input generation using *imperative* constraints, i.e., constraints written as predicates in an imperative language. We built upon Korat, a technique for solving such constraints using execution-driven monitoring, where the given predicate is executed on candidate inputs to filter and prune invalid inputs and to generate valid ones. We used the insight that a static lightweight data-flow analysis of the given imperative constraint can enable more efficient solving. We described an approach that embodies our insight and evaluates it using a suite of well-studied subject constraints. The experimental results show that our approach provides a substantial speedup over previous work.

Acknowledgments

This material is based upon work partially supported by the Fulbright program, the National Science Foundation under Grant Nos. IIS-0438967, CCF-0746856, and CCF-0845628, and AFOSR grant FA9550-09-1-0351.

References

[1] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke, "LLVA: A Low-level Virtual Instruction Set Architecture," in *Proc. 36th MICRO*, 2003, pp. 205–216.

[2] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing based on Java Predicates," in *Proc. ISSTA*, 2002.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," in *Proc. 13th CCS*, 2006, pp. 322–335.

[4] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," *Proc. TACAS*, pp. 168–176, 2004.

[5] L. A. Clarke, "Test Data Generation and Symbolic Execution of Programs as an aid to Program Validation." Ph.D. dissertation, University of Colorado at Boulder, 1976.

[6] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Trans. Softw. Eng.*, vol. 23, pp. 437–444, July 1997.

[7] P. T. Darga and C. Boyapati, "Efficient Software Model Checking of Data Structure Properties," in *Proc. 21st OOPSLA*, 2006, pp. 363–382.

[8] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley, "STARC: Static Analysis for Efficient Repair of Complex Data," in *Proc. 22nd OOPSLA*, 2007, pp. 387–404.

[9] B. Elkarablieh, D. Marinov, and S. Khurshid, "Efficient Solving of Structural Constraints," in *Proc. ISSTA*, 2008.

[10] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proc. ICSE*, 2010.

[11] M. Gligoric, T. Gvero, S. Lauterburg, D. Marinov, and S. Khurshid, "Optimizing Generation of Object Graphs in Java PathFinder," in *Proc. 2nd ICST*, 2009, pp. 51–60.

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *Proc. PLDI*, 2005, pp. 213–223.

[13] J. Huang, "An Approach to Program Testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, Sep. 1975.

[14] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Trans. Software Engg. and Methodology*, vol. 11, no. 2, pp. 256–290, Apr. 2002.

[15] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[16] S. Khurshid, C. S. Pasareanu, and W. Visser, "Generalized Symbolic Execution for Model Checking and Testing," in *Proc. 9th TACAS*, 2003, pp. 553–568.

[17] J. C. King, "Symbolic Execution and Program Testing," *Communications ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[18] D. Marinov, "Automatic Testing of Software with Structurally Complex Inputs," Ph.D. dissertation, Massachusetts Institute of Technology, 2005.

[19] D. Marinov and S. Khurshid, "TestEra: A Novel Framework for Automated Testing of Java Programs," in *Proc. 16th ASE*, 2001, pp. 22–31.

[20] S. Misailovic, A. Milicevic, N. Petrovic, S. Khurshid, and D. Marinov, "Parallel Test Generation and Execution with Korat," in *Proc. 6th ESEC/FSE*, 2007, pp. 135–144.

[21] C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen, "On the Automated Generation of Program Test Data," *IEEE Trans. Software Engg.*, vol. 2, no. 4, pp. 293–300, Jul. 1976.

[22] D. A. Ramos and D. R. Engler, "Practical, Low-Effort Equivalence Verification of Real Code," in *Proc. 23rd CAV*, 2011, pp. 669–685.

[23] K. Sen, D. Marinov, and G. Agha, "CUTE: A Concolic Unit Testing Engine for C," in *Proc. 5th ESEC/FSE*, 2005, pp. 263–272.

[24] J. H. Siddiqui and S. Khurshid, "An Empirical Study of Structural Constraint Solving Techniques," in *Proc. ICFEM*, 2009, pp. 88–106.

[25] J. H. Siddiqui and S. Khurshid, "PKorat: Parallel Generation of Structurally Complex Test Inputs," in *Proc. 2nd ICST*, 2009, pp. 250–259.

[26] J. H. Siddiqui, D. Marinov, and S. Khurshid, "Optimizing a Structural Constraint Solver for Efficient Software Checking," in *Proc. 24th ASE*, 2009.

[27] K. Stobie, "Model Based Testing in Practice at Microsoft," *Electronic Notes in Theoretical Computer Science*, vol. 111, pp. 5–12, 2005.

[28] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson, "Software Assurance by Bounded Exhaustive Testing," in *Proc. ISSTA*, 2004, pp. 133–142.

[29] W. Visser, K. Havelund, G. Brat, and S. Park, "Model Checking Programs," in *Proc. 15th ASE*, 2000, p. 3.